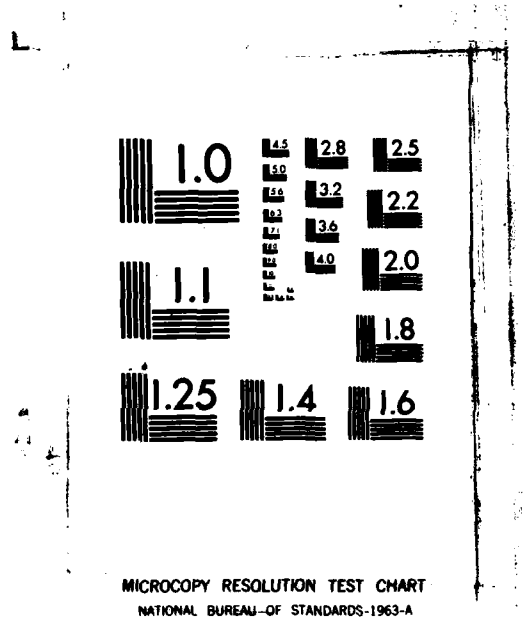
AD-A140 852 THE PRINGLE PARALLEL COMPUTER(U) PURDUE UNIV LAFAYETTE 1/1
IN DEPT OF COMPUTER SCIENCES A A KAPAUAU ET AL. APR 84
TR-84-04-01 N00014-80-K-0816

UNCLASSIFIED F/G 9/2 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
8.0
9.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

BLUE CHiP

(12)

# The Pringle Parallel Computer

by

Alejandro A. Kapauan
J. Timothy Field
Dennis B. Gannon
Lawrence Snyder





**The BLUE CHiP Project**
University of Washington
Department of Computer Science, FR-35
Seattle, Washington 98195

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

| REPORT DOCUMENTATION PAGE | | READ INSTRUCTIONS BEFORE COMPLETING FORM |
|---|---|---|
| 1. REPORT NUMBER<br>TR-84-04-01 | 2. GOVT ACCESSION NO.<br>A140 852 | 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE (and Subtitle)<br>THE PRINGLE PARALLEL COMPUTER | | 5. TYPE OF REPORT & PERIOD COVERED<br>Technical, interim<br>6. PERFORMING ORG. REPORT NUMBER |
| 7. AUTHOR(s)<br>Alejandro A. Kapauan, J. Timothy Field, Dennis B. Gannon, Lawrence Snyder | | 8. CONTRACT OR GRANT NUMBER(s)<br>N00014-80-K-0816<br>N00014-81-K-0360<br>N00014-84-K-0143 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS<br>University of Washington<br>Department of Computer Science, FR-35<br>Seattle, Washington 98195 | | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS<br>Task SRO-100 |
| 11. CONTROLLING OFFICE NAME AND ADDRESS<br>Office of Naval Research<br>Information Systems Program<br>Arlington, Virginia 22217 | | 12. REPORT DATE<br>April 1984<br>13. NUMBER OF PAGES<br>9 |
| 14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) | | 15. SECURITY CLASS. (of this report)<br>Unclassified<br>15a. DECLASSIFICATION/DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this report is unlimited.

DISTRIBUTION STATEMENT A
Approved for public release
Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

DTIC ELECTE S MAY 7 1984 D B

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

highly parallel computer, Pringle, CHiP Computer, wisdom-of-experience problem, Switch, polled-bus, parallel architecture

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

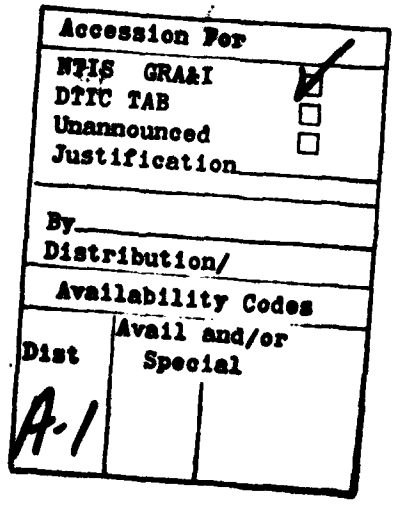
The Pringle is a 64 processor MIMD computer with a 64 M (8 bit) instructions per second execution rate. (Copies are running at Purdue and Washington). The Pringle runs programs written for the Configurable, Highly Parallel (CHiP) Computer. That is, the Pringle executes the 64 separate instruction streams as well as the interconnection (phase) stream that configures the lattice of a 64 processor CHiP computer. But the Pringle is not a CHiP. It gives the illusion of the CHiP machine's conflict-free, point-to-point communication

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

using a 64 Mbit internal polled bus. In addition to describing the design goals and the Pringle architecture, this paper identifies two problems common to novel architecture implementation projects, the "wisdom-of-experience" problem and the "single-instance" problem, and explains how they were addressed for the Pringle.



# The Pringle Parallel Computer

Alejandro A. Kapauan
J. Timothy Field
Dennis B. Gannon

Purdue University

Lawrence Snyder

University of Washington

TR-84-04-01

# THE PRINGLE PARALLEL COMPUTER

*Alejandro Kapauan*
*J. Timothy Field*
*Dennis B. Gannon*

**Purdue University**

*Lawrence Snyder*

**University of Washington**

## Abstract

**The Pringle is a 64 processor MIMD computer with a 64 M (8 bit) instructions per second execution rate. (Copies are running at Purdue and Washington.) The Pringle runs programs written for the Configurable, Highly Parallel (CHiP) Computer. That is, the Pringle executes the 64 separate instruction streams as well as the interconnection (phase) stream that configures the lattice of a 64 processor CHiP computer. But the Pringle is not a CHiP. It gives the illusion of the CHiP machine's conflict-free, point-to-point communication using a 64 Mbit internal polled bus. In addition to describing the design goals and the Pringle architecture, this paper identifies two problems common to novel architecture implementation projects, the "wisdom of experience" problem and the "single instance" problem, and explains how they were addressed for the Pringle.**

## 1. Introduction

**Computer architects, especially those concerned with parallel computers, are often criticized for only producing paper designs, for never implementing their ideas. In their defense, however, there are two problems with implementing a computer design besides the obvious one of having to expend enormous amounts of time and money:**

> **(i) *Wisdom-of-experience problem.* Even after extensive simulation, making the correct design choices often requires the experience of having used the machine extensively – one must have one to build one.**
>
> **(ii) *Single-instance problem.* The construction of a computer demonstrates only one set of design choices – all other design alternatives remain unexplored.**

**So architects risk their time and money on a computer implementation that, by (i), involves certain design decisions they don't have the experience to make and that, by (ii), may be inconclusive in terms of revealing what the proper design decisions should have been. No wonder very few novel architectures are actually implemented.**

**The Pringle is a 64 processor MIMD architecture that *has* been implemented. It runs programs written for the Configurable, Highly Parallel (CHiP) Computer.[1,2] The Pringle avoids problem (i) and solves problem (ii) for the CHiP* machine. To explain, consider the following.**

**The CHiP Computer is a highly parallel architecture that has been designed to exploit VLSI implementation. It is especially sensitive to the wisdom-of-experience problem: Each chip of the CHiP Computer will contain processor(s), memory and switching facilities. This forces all major system components to compete for the same resource, silicon. Since there are significant trade-offs (described in detail below), striking a balance among those competing components requires extensive experience running CHiP Computer programs. Software emulation is quickly overwhelmed, so, a computer to execute CHiP programs is needed in order to design the CHiP Computer.**

**The work described herein is part of the Blue CHiP Project which has been funded in part by the Office of Naval Research under Contracts N00014-80-K-0816, N00014-81-K-0360 (Special Research Opportunities Task SRO-109), and N00014-84-K-0143.**

*A brief description of the CHiP architecture is given in Appendix A.

The Pringle is such a computer. It avoids (to a large degree) the wisdom of experience problem because the Pringle is not a CHiP. It implements the CHiP architecture in such a way that questions such as balancing the use of silicon among competitors do not arise.* But it does even more. The Pringle solves the *single-instance problem* because it provides direct hardware emulation of an interesting variety of CHiP architectures reflecting many combinations of design choices. Thus, the Pringle not only gives us a parallel computer on which to run CHiP programs, it does so in a way that does not require uninformed design choices, and does permit an exploration of the design space.

The Pringle appears to stack up as a better computer than the CHiP, but it is not really better, just different. The key to avoiding problem (i) and solving problem (ii) is a 64 Mbit/sec. polled bus, called the Switch (see Figure 1 and Table 1). But this structure will not generalize to arbitrarily many processors, while the lattice of the CHiP architecture will. Nevertheless, the Pringle architecture has intrinsic interest, and serves as a case study of how a preprototype can deflect the wisdom-of-experience and the single-instance problems for a new machine.

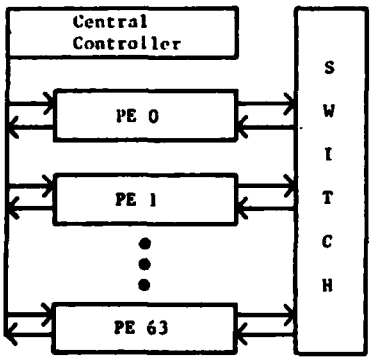



Figure 1. Schematic diagram of the Pringle showing the processing elements (PEs) and the Switch, implemented as a polled bus. (See Figure 3 for more detail.)

*This statement is true not simply because the Pringle was implemented in TTL; it is true for architectural reasons that are described in Section 2.

The purpose of this paper is to describe the Pringle architecture and to explain its role as a surrogate for the CHiP Computer. The next section gives the design goals and an overview of the Pringle. Section 3 gives the detailed design of the machine. The final section discusses the Pringle in terms of the two problems mentioned above. Specifically, it points out those few places where the Pringle did not avoid the wisdom of experience problem, and explains which architectural features solve the single instance problem by supporting exploration of the design space.

**The Pringle Engine Data Sheet**

| | |
|---|---|
| Number of PEs | 64 |
| PE microprocessor chip | Intel 8031 |
| PE data path width | 8 bits |
| PE floating point chip | Intel 8231 |
| PE RAM size | 2K bytes |
| PE EPROM size | 4K bytes |
| PE clock rate | 12 MHz |
| Switch structure | polled bus |
| Switch clock rate | 8 MHz |
| Bus bandwidth | 64 Mbits |
| Switch data path width | 8 bits |
| Controller | Intel 8086 |
| Processing rate | 64 Mips |

Table 1. Pringle data sheet.

## 2. Pringle Overview

Before presenting the structure of the Pringle, a few comments are in order about specific design goals.

Goal 1: Provide a "good first approximation" to the CHiP architecture, but with enough flexibility to extend or limit the facilities to reflect design alternatives.

This is the goal alluded to above. Here "good first approximation" refers to a best guess as to how the silicon area of a CHiP machine will be split among processors, the memories and the switch facilities. To see the issues, suppose that with current technology one can place a processor, some random access memory and a few switches on one chip. Since

the chip's size is essentially fixed, changing one constituent is done at the expense or advantage of the other two (Figure 2). More of one implies less of the others. But there are other considerations besides simple capacity: More switches give greater programming convenience at the expense of added propagation delay; a gross imbalance in processor speed compared with PE-to-PE communication speed affects the efficiency of many algorithms. So, balancing the usage is a delicate matter. How it was handled will be described in Section 4.

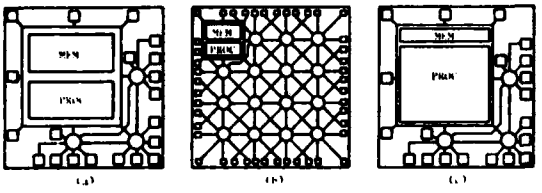


Figure 2. Schematic diagrams of VLSI chip alternatives for constructing a CHiP lattice: (a) balanced, (b) much switching, (c) large processor; small squares represent bonding pads and circles represent switches.

Goal 2: Provide "many" processing elements.

A second consideration was that the Pringle must have enough processors to test adequately the fine-grain parallelism characteristic of CHiP processing. Of course, no parallel processor has enough processing capacity to handle the largest problems of interest, so we must in any case address the issues of contracting large problems and multiplexing the processors. But there should be enough capacity to observe sustained performance on nontrivial problems.

Goal 3: Support multiple I/O protocols.

Another feature of the Pringle design is that it permits a comparison of data driven I/O with synchronous I/O. Data driven communication is expensive to implement because of the need for components like input queues and "overrun" signalling mechanisms. On the other hand, synchronous I/O, which requires PEs to communicate only at agreed upon times, is difficult to program, potentially fragile, but possibly faster. It has been shown[3] that certain data driven programs can be automatically converted into equivalent, synchronously communicating programs. It is crucial to be able to run both to determine the effect on performance.

Goal 4: Balance the processing and data transfer rates.

In order for the Pringle to provide useful performance data, the execution speeds of its various components must be balanced, *i.e.* the ratio of the speeds of any two components must be the same as it is likely to be in the VLSI implementation. Said in a more derogatory way, the components must all be slow by the same factor.

With these goals in mind, we can consider the overall structure of the Pringle.

The system is divided into two distinct logical parts (see Figure 3). The first is a processing element array controlled by a central microprocessor. It contains 64 PEs each of which has its own read-write random access memory and read-only memory. The processors of these PEs are 8-bit single-chip microcomputers coupled with arithmetic processing units (APUs) which perform 32-bit floating point arithmetic.

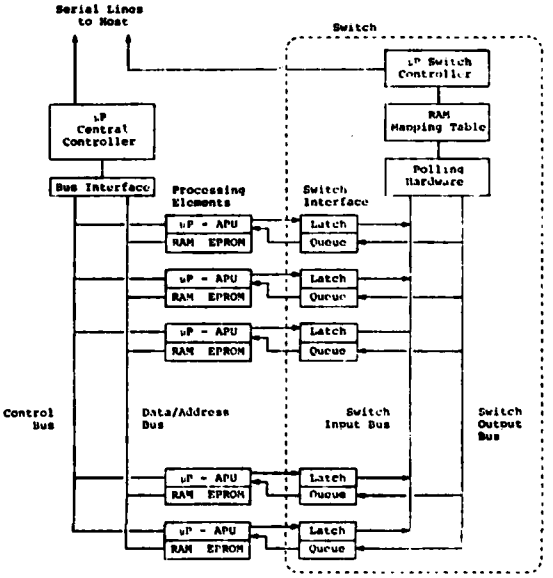



Figure 3. The Pringle architecture.

The PE array is managed by a controller, a 16-bit microprocessor which communicates with the PEs by means of an address-data bus, and a control bus. To facilitate quick down loading of data and programs into PE RAMs from the controller, the RAM of each PE is made to appear as a block of memory in the address space of the controller's microprocessor. This *memory mapping* allows the controller to examine all PE RAM and to take snapshots of memory during the execution of a CHiP program. The control bus includes global reset and interrupt lines which permit the controller to halt or pause the PEs, and status lines which allow the controller to determine the busy or idle status of the PEs.

The second part of the system is the emulator for the CHiP's lattice, which we shall call the Switch. Recall (e.g. from Appendix A) that the lattice provides direct point-to-point communication for the PEs. There are no conflicts and all PEs can be performing I/O simultaneously. The Switch implements this structure with a polled bus. A mapping table gives the specific routing information. The table contains source-destination pairs of the form:

Table[sourcePEno,portno]=(destinationPEno,portno)

It uses the algorithm given in Figure 4.

Every PE is interfaced to the Switch with an output data latch and an input data queue. Assuming processing elements with 8 ports as shown in Figure A, we can assign direction addresses 0 through 7 to the PE I/O ports. When a PE wants to write to an I/O port in Pringle, it latches both the data and the number of the desired direction onto its output data latch, setting a data-present flag on the latch. The Switch polling hardware does a cyclic scan of all the output data latches via the Switch input bus. (See the algorithm in Figure 4.) When it encounters a latch with the data-present flag set, it takes the data and direction number from the latch, clears the data present flag, then looks up in the Table in high-speed RAM for the destination PE and port number for the data. The polling hardware then routes the data to the input queue of the destination PE with the destination port number appended to the data, via the Switch output bus. The polling hardware will run at a maximum speed of 8 MHz, allowing a complete 64 PE scan to take place in 8 microseconds. Although not exceptionally fast, this speed is consistent with the computational rate of the PEs, i.e. it is balanced as explained in Goal 4.

```
L:  s ⟵ s+1                          /* source PE index */
    if PE[s].datapresent
    then {PE[s].datapresent ⟵ FALSE   /* clear flag */
          temp ⟵ PE[s].data;          /* save data to be transmitted */
          d ⟵ Table[s,PE[s].portno].destinationPEno
                                      /* get distination location */
          p ⟵ Table[s,PE[s].portno].portno
          PE[d].dataQ ⟵ temp          /* place data on Queue */
          PE[d].portQ ⟵ p};           /* place port number on Queue */
    go to L;
```

Figure 4. Polling algorithm for the Switch. Each output latch has a data present flag, three bits of port number and eight bits of data. Each input queue frame has eight bits of data and three bits of port number.

The RAM which contains the Switch mapping table is accessible to a microprocessor which serves as the Switch controller. It can down load switch settings into the RAM, it can halt and start the polling hardware, and it can detect abnormal conditions in the Switch hardware such as an input queue overflow at one of the PEs.

There is sufficient memory space in the mapping table RAM to hold 8 different configuration settings at the same time. This allows up to 8 different interconnection structures to be down loaded into the Switch at once. The Switch controller can select any one of the eight configuration settings even as the polling hardware is running.

## 3. Pringle Architecture Specifics

In this section we explain the design at a more detailed level. It is possible to skip this section on first reading.

### 3.1 Processor Elements

Figure 5 presents a block diagram of the PEs implemented in the Pringle machine. (Note that these are similar to the PEs in the NOSC systolic array.[6]) The microprocessor used is an Intel 8031, a single-chip 8-bit microcomputer. It contains 128 bytes of internal read-write memory, two parallel I/O ports, two counter-timers, and a serial I/O port. It runs on a 12 MHz clock which gives it a 1 microsecond execution time for most of its instructions, and a maximum instruction execution time of 4 microseconds for 8-bit multiply and divide.

External memory is composed of an industry

standard 2048 by 8-bit static RAM and a 4096 by 8-bit EPROM. A simple system of tri-state buffers allows the central controller to access the external RAM when it is not being accessed by the 8031.

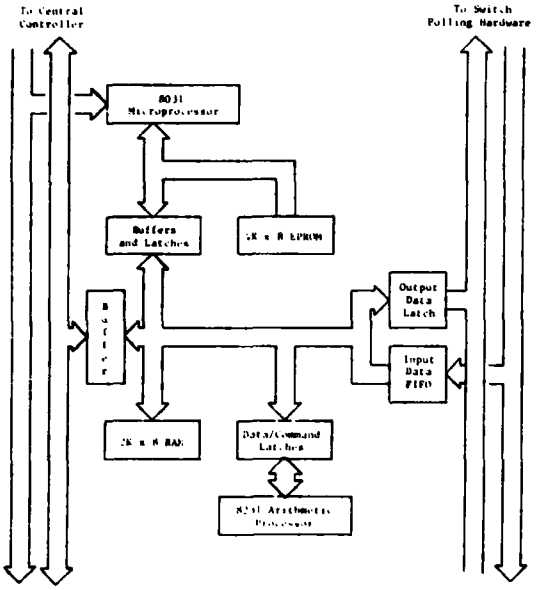



Figure 5. PE detail

An Intel 8231 arithmetic processing unit (APU) chip is interfaced to the 8031 by means of a command latch and a data latch. The 8231 contains its own stack to which the 8031 can push data, and from which it can pop data. Commands may be issued by the 8031 to the APU to make it perform floating point arithmetic operations on the stack's contents. As the APU executes commands, the 8031 microprocessor is free to perform other operations.

The 8031 has access to an 11 bit wide output data latch and an 11 bit wide input data queue which, as mentioned earlier, interface it to the switch lattice emulator. Eight of these bits are the data sent to or received from the Switch, while the other three specify the port direction. Since the microprocessor data bus is only 8 bits wide, three of the microprocessor parallel I/O port lines serve to extend the data path width to 11 bits. Other I/O port lines serve as control signals to the latch and queue.

The input queue acts as a buffer between the Switch hardware, which can operate at a burst data rate of up to 8 MHz, and the relatively slower PE microprocessor. Since all eight logical input ports to the PE are implemented as one physical input port, the use of a buffer queue is essential. The buffer used consists of three bipolar FIFOs, 4 bits wide by 16 deep, which produces a single 16 byte queue. Assuming that the PEs transmit 32 bit words of data, the resulting queue is capable of holding up to four words. This implies that sufficient buffering is present to allow the emulation of CHiP machine programs wherein up to four PEs write to a single PE in a single CHiP machine cycle.

### 3.2 Switch Emulator Structure Detail

The Switch is implemented in Schottky TTL hardware to permit a very fast clock rate to be used. Figure 6 presents the block diagram of the polling circuitry, mapping table and Switch controller.

A six bit polling counter is used to cycle the input address bus through the addresses of all 64 PEs. When a PE is addressed, it responds by putting the

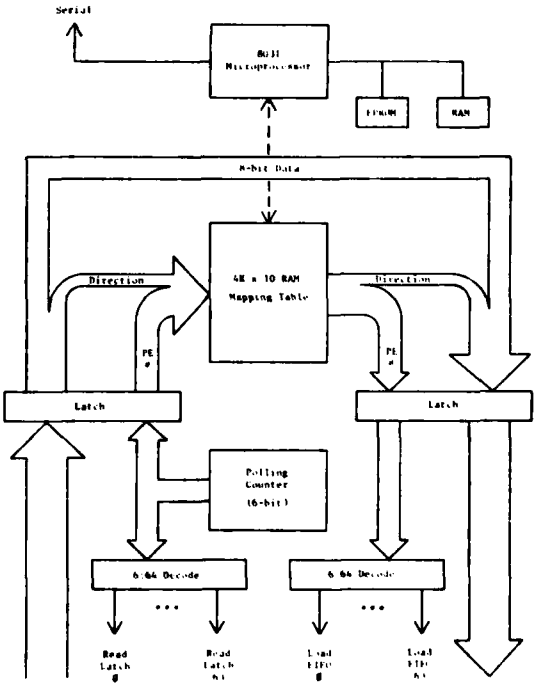



Figure 6. Switch detail

status of its data present flag on the data present bus line, and the contents of its output latch on the data bus. When the hardware detects a latch which contains data, it sends a strobe pulse on a bus control line which clears the data present flag of the currently addressed PE, and latches the data on the bus. Using the PE number from the polling counter concatenated with the direction number supplied by the PE, the hardware looks up the mapping table RAM for the destination PE number and direction number.

The destination PE number is latched on the output address bus, and the data from the source PE and the destination port address are latched on the output data bus. Then a strobe pulse is sent on an output control line. This causes the contents of the data bus to be entered in the queue of the selected PE.

The entire operation is pipelined to allow the polling of the next input data latch to take place while the data from the current PE is routed to its destination. Notice that this scheme was designed only to emulate a switch lattice for a limited number of PEs. It cannot replace a true switch lattice for an arbitrarily large number of PEs because of the inherent serial bottleneck in sequential polling.

An 8031 microcomputer with 4096 bytes of program EPROM and 2048 bytes of scratch pad RAM serves as the controller of the switch. It can stop the clock on the polling hardware and read or write to the mapping table RAM. By means of three control lines, it can specify which of the 8 different switch settings is active at a given instant of time. A serial line allows the 8031 to communicate with the host system.

The mapping table resides in a 4096 by 10-bit word RAM. For each configuration setting, each PE requires 8 words, one for the destination PE number and port number of its eight output ports. Thus a total of 512 words are needed per switch setting, giving room for eight different configurations.

### 3.3 Physical Characteristics

Excluding power supplies, Pringle occupies three 10.5 inch high cages on a standard 19 inch wide rack. Wire-wrap boards are used, 9.6 by 7.8 inches in size, to make hardware modification easy.

There are sixteen PE boards, each containing a cluster of four PEs, and eight Switch boards, each containing the data latches and queues for eight PEs. In addition there is a Switch controller board containing the polling hardware and control microprocessor for the Switch, and a bus interface board which allows the 8086 central controller to communicate with all the PEs.

Each PE used 22 ICs; the entire machine, including the Switch, contains 1947 ICs.

### 3.4 External Input/Output System

The external input/output (XIO) system has been designed and is under construction; we give only a brief overview. Figure 7 gives the logical picture of the XIO system.

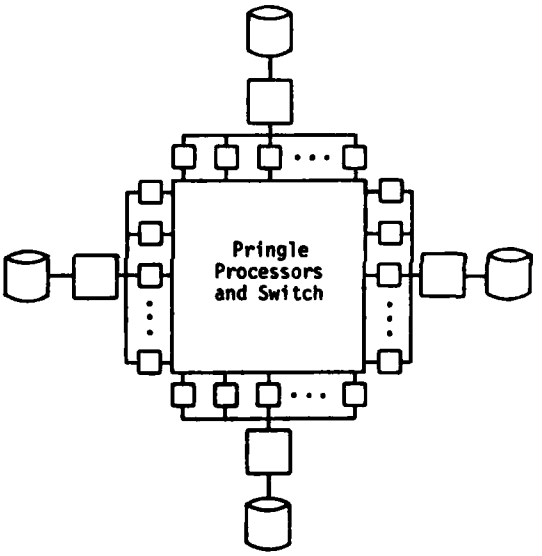



Figure 7. The logical structure of the external input/output system.

Data is stored on one of four Winchester disks. The file system for the disks is managed by the attached Intel 8086 microprocessor. These machines split files apart into streams for input to the pringle engine or they take streams output by the engine and join them together to form files. The streams enter or leave the engine through one of eight buffers which dampens out transfer bursts and allows the engine to maintain maximum speed. Each buffer is

composed of 4K bytes of storage, an Intel 8031 microprocessor controller, and a 4K byte EPROM. The memory of each buffer is divided into two 2K segments. Exclusive access to a segment of the memory is provided to either the buffer controller or the file manager; there are no conflicts in referencing this memory.

In the logical picture the buffers are attached directly to the switches. In the physical picture, the buffer elements are interfaced to the Switch in essentially the same way that the PEs are, i.e. the 32 buffer elements are polled just like the PEs.

## 4. Comparison of the Pringle and the CHiP Computers

The Pringle has been described in sufficient detail to permit a discussion of its role in avoiding the wisdom-of-experience and the single-instance problems.

As mentioned above, a CHiP Computer uses the silicon for processors, memories and switches. The problem of selecting a balance between these three uses of silicon was largely by-passed in the Pringle. Specifically, the switch facilities were separated from the PEs and implemented as the polled bus. The memory was moved from the processor chip to a separate memory chip, a natural choice for a non-VLSI implementation. But more importantly, the memory mapping facility in the controller can support paging (in an experimental context) thus giving a larger logical memory than physical memory and permitting experimentation with different memory sizes. The processor was augmented with a substantial EPROM to permit enhancement of the instruction set. This has been used by Gannon and Wang[5] to implement an interpreter for a stack machine in the EPROM as an alternative to the originally proposed microprocessor instruction set. The conclusion is that by separating the three pieces we were able not only to avoid striking a balance among the three components, we were able to include features that made experimentation convenient.

We now consider an important way that the single-instance problem was solved.

Perhaps the most crucial characteristic of a CHiP Computer's switch lattice is the *corridor width*, the number of switches separating two adjacent PEs. (Figure A shows two lattices with corridors of width one and two respectively.) The corridor width is a fixed characteristic of a machine, though different family members have different widths. Wide corridors provide greater data routing capability for complex topologies, and although any topology can be embedded into any suitably large lattice, those with narrow corridors may underutilize the PEs as a consequence.[1,2] Wide corridors are convenient. On the cost side of the ledger, wide corridors require many switches and data paths, and a reduced proportion of silicon is devoted to processor and memory capacity. Moreover, there is an increased pin requirement per package with wide corridors, and a (minor) increase in transmission delay for neighborhood communication. Like all architectural features, the appropriate corridor width is determined by the needs of typical algorithms.

The Pringle's Switch can implement lattices with any corridor width. The key to this ability rests in the fact that regardless of corridor width, the lattice generally*** implements point-to-point communication paths. Thus, a lattice of any corridor width, once reduced to a set of point-to-point communication paths, can be "implemented" by down loading the source-target pairs into the Switch's mapping table.

The routing constraints, imposed on the programmer by a lattice with a particular corridor width, are enforced by the Poker Parallel Programming Environment,[4] the Pringle's front end software system. There the programmer specifies the lattice he wishes to use, programs the interconnection structure graphically, and "compiles" the result into source-target pairs. In Poker it is NOT possible to violate the limitations of the selected corridor width, so the distilled communication description received by the Pringle is always a fair rendering of the routing capability of that lattice. As a result the Pringle looks to the programmer like a CHiP Computer with the lattice of his choice.

In summary, the Pringle can run CHiP programs, but it doesn't so much look like a CHiP Computer as it looks like a family of CHiP Computers.

---

***A CHiP switch can fan-out, i.e., broadcast, but this feature can be realized by other means and has been of only limited utility so far. The Switch cannot broadcast.

## Acknowledgements

It is a pleasure to thank J.J. Symanski of the Naval Ocean Systems Center for his guidance and support. John May provided tireless technical help on the first copy of Pringle and constructed the second copy single-handedly. Staff of the Purdue Computer Center, especially Lonnie Ahlen, Art De Armond and John Steele have supported us with equipment and advice. Intel Corporation generously donated the processors, memories and floating point chips. We are grateful for all of the assistance.

## References

[1] Lawrence Snyder
Introduction to the Configurable, Highly Parallel Computer
*Computer*, 15(1): 47-56, January 1982

[2] Lawrence Snyder
Overview of the CHiP Computer
In John P. Gray, editor, *VLSI 81*, Academic Press, pages 237-246, 1981.

[3] J.E. Cuny and L. Snyder
Compilation of Data-driven Programs for Synchronous Execution
Proceedings of the 10th Principles of Programming Languages, ACM, pages 197-202, 1983

[4] Lawrence Snyder
Introduction to the Poker Parallel Programming Environment
Proceedings of the International Conference on Parallel Processing, IEEE, pages 289-292, 1983.

[5] Dennis B. Gannon and Ko-Yang Wang, unpublished notes.

[6] Keith Bromley, J.J. Symanski, J.M. Speiser and H.J. Whitehouse
Systolic Array Processor Developments
In H.T. Kung, Bob Sproull and Guy Steele, editors, *VLSI System and Computations*, Computer Science Press, pages 273-284, 1981.

## Appendix A: Brief Description of the CHiP Computer

The CHiP Computer[1,2] is composed of a collection of homogeneous processing elements (PEs) placed at regular intervals in a lattice of programmable switches (see Figure A). Each PE is a simple microprocessor with a small amount (e.g., 2K bytes) of local memory for program and data storage; there is *no global memory*. Each switch contains a small amount (e.g., 8-16 words) of memory in which to store switch instructions, called *configuration settings*. Executing a configuration setting causes a switch to connect two or more of its incident data paths; note that this is circuit switching. Separate data paths can cross the switch simultaneously (i.e., there is crossover at a switch). By programming the switches appropriately, the PEs can be connected into the topologies of arbitrary form, e.g., mesh, tree, torus (see Figure B).

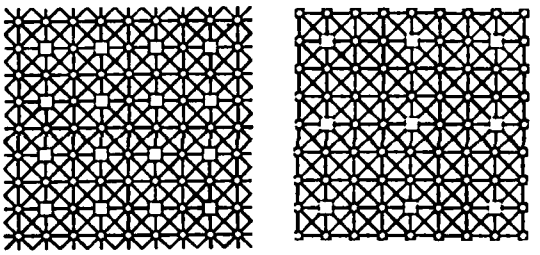


Figure A. Two switch lattices; squares represent PEs, circles represent switches, lines represent data paths; PEs are actually much larger than switches.

In addition to the switch lattice, a CHiP architecture has a controlling computer responsible for monitoring the computation. The computation is divided into *phases*, where each phase corresponds roughly to a single algorithm with a single processor topology. For example the first phase might be a mesh connected phase, the second a tree connected phase, etc. The controller prepares for a computation by down loading to the PEs the code segments needed for several phases, and down loading to the switches the configuration settings implementing the topologies of those phases. To initiate computation, the controller broadcasts to the switches the address

(in the switches) of the configuration setting for the topology required for the first phase. This causes the processors to be connected into that topological structure. The PEs then begin executing their respective code segments for that phase using a common clock. PEs simply read and write to their I/O ports without "knowing" the source or target PEs of the transfer; the data paths of the configuration form point-to-point connections. When the phase is complete, the controller broadcasts a signal indicating which configuration setting is needed for the next phase, and the PEs then begin executing their corresponding code segments. Execution continues in this manner until the computation is complete or until additional PE and switch codes have to be down loaded.

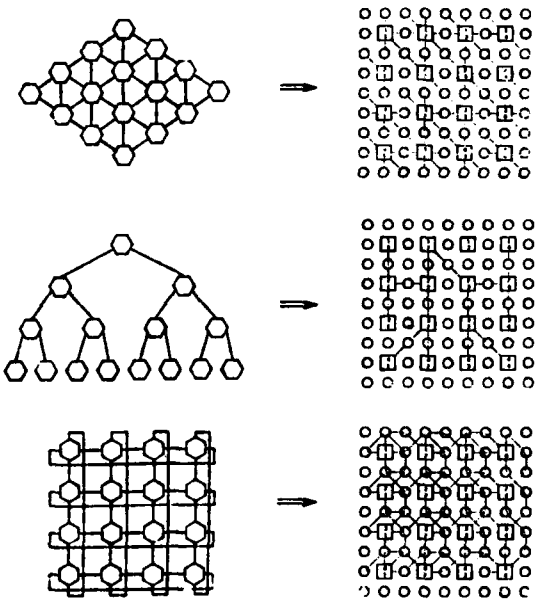
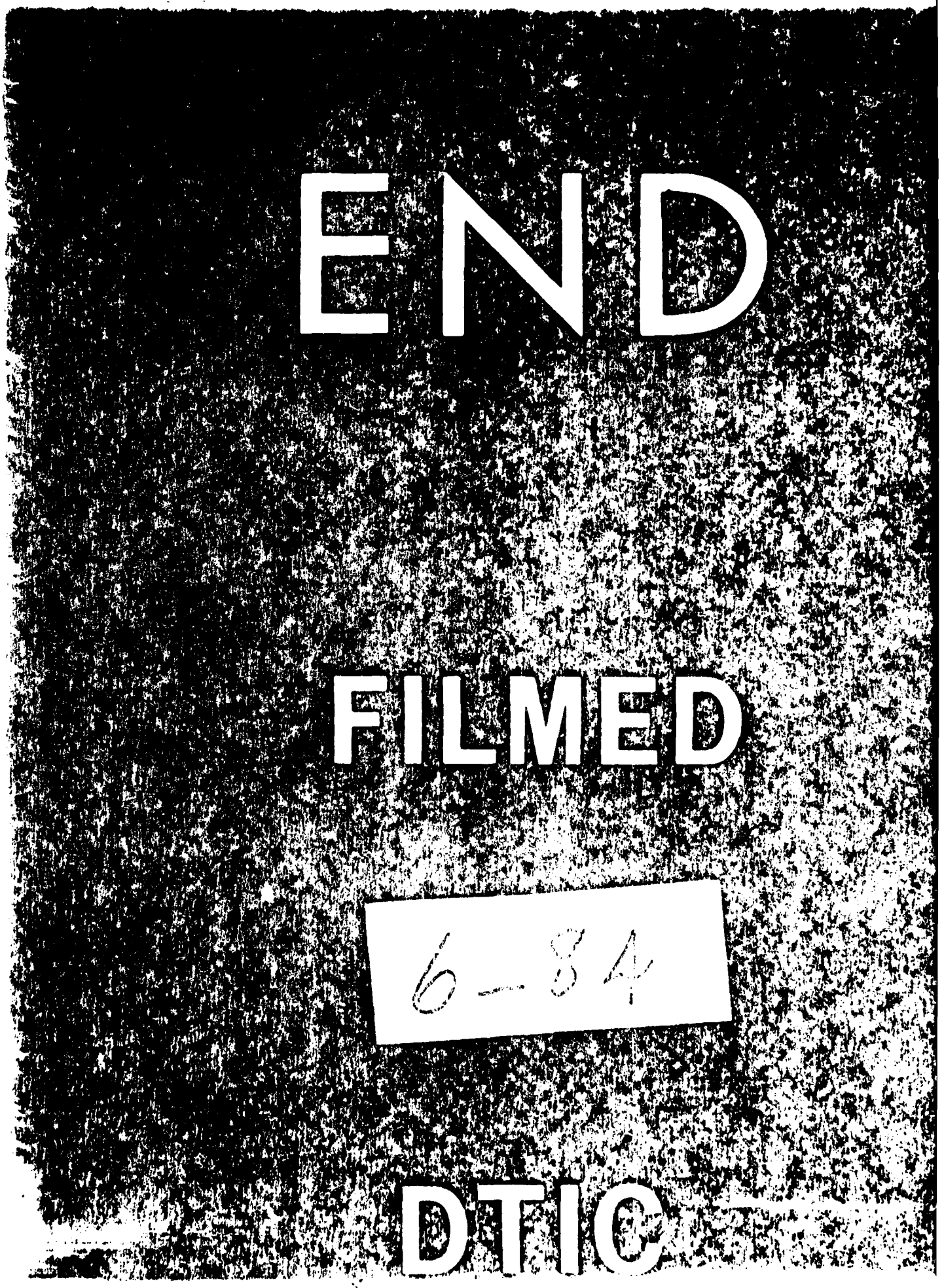


*Figure B. Three configurations of the lattice of Figure A.*


END
FILMED
6-84
DTIC